Advanced Parallel Programming

Mesh Decomposition: Basic Concepts and Decomposition Algorithms

David Henty
EPCC, University of Edinburgh
d.henty@epcc.ed.ac.uk

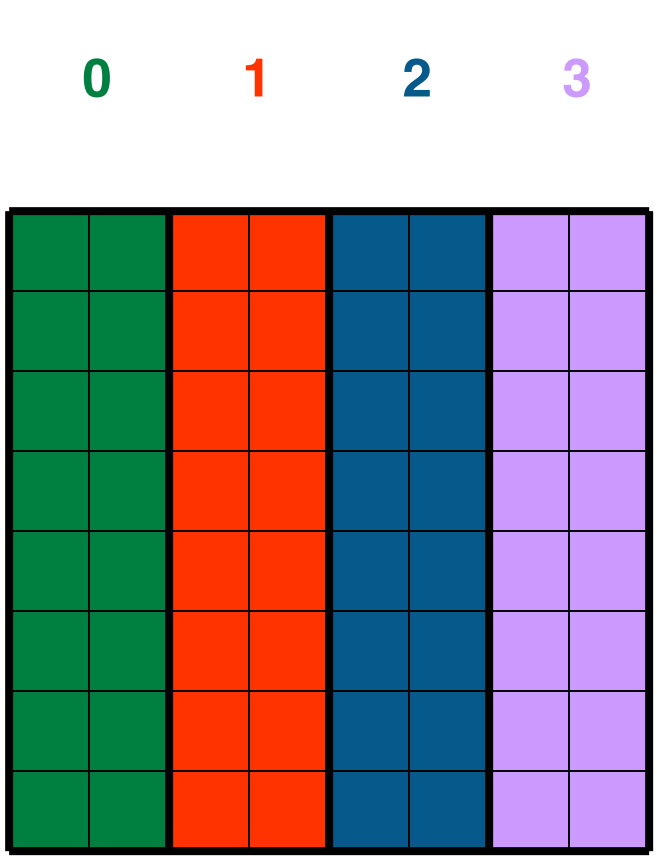
Structured Meshes

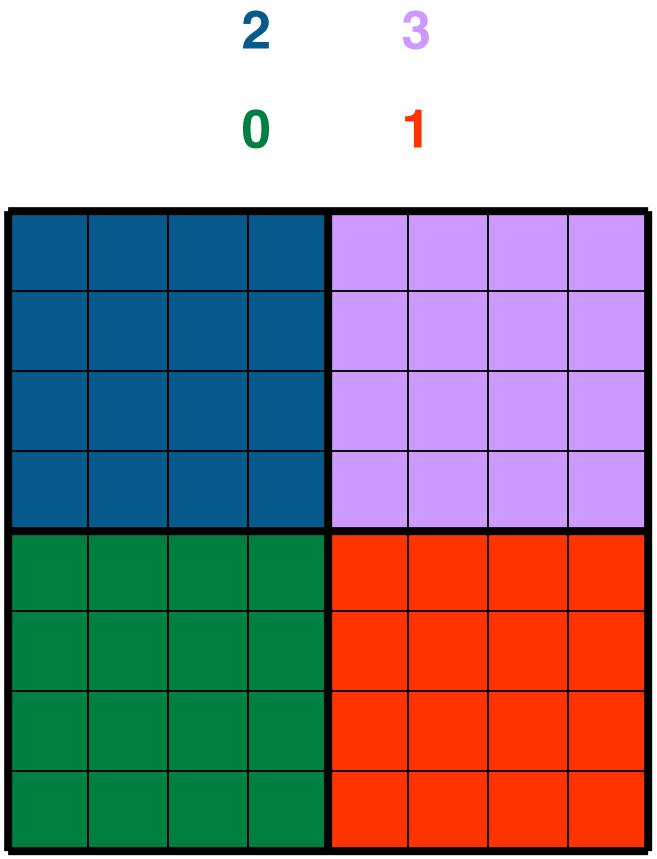
- Many problems can be solved on a regular grid
  - eg Game of Life, Image Processing, Predator-Prey model, ...
  - regular grid is also called a Structured Mesh
- When we decompose the problem domain
  - aim for load balance across processors
  - with a minimum amount of communication
- Load balance is an equal number of cells on each processor
  - ie each subdomain must have the same **area** (2D) or **volume** (3D)
- If each cell depends on its nearest neighbours
  - comms happens when neighbouring cells are on different processors
  - want to minimise the **length** of the subdomain **boundaries** (2D)
    - or the **area** of the subdomain's **surface** (3D)

Example

- Test problem
  - each cell depends on four nearest neighbours (no diagonals)
  - periodic boundary conditions
  - look at an 8x8 simulation on 4 processors
- How are the load balance and communications costs affected by different decompositions?
  - speed of calculation limited by size of largest subdomain
  - communications cost is related to the size of the boundaries
- NOTE
  - real simulations would be **MUCH LARGER** than 8x8!

1D and 2D Decompositions

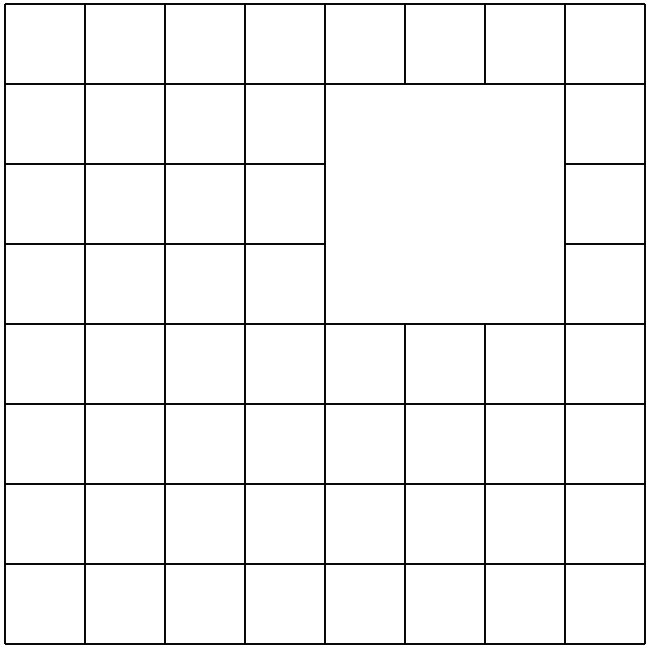


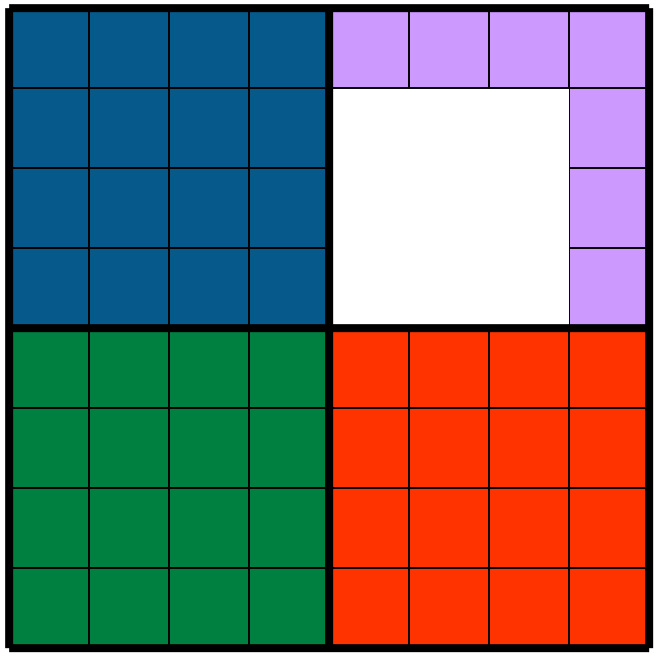


- load: 16,16,16,16
- boundary: 16+16+16+16=64

- **load: 16,16,16,16**
- **boundary: 12+12+12+12=48**

Load-imbalanced Problem



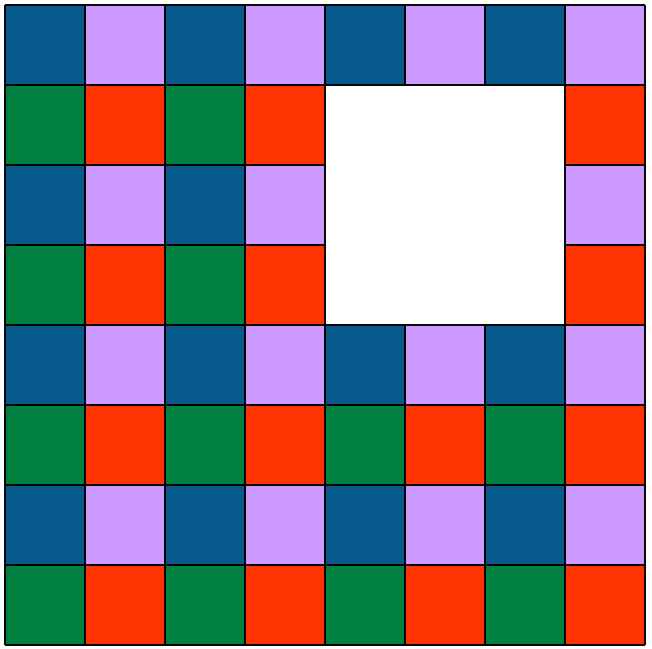


- **Mesh with a hole**
  - **a 3x3 area with no calculation**

- Regular decomposition
  - load: 16,16,16,7
  - boundary: 12+10+10+7=39

Cyclic Distribution

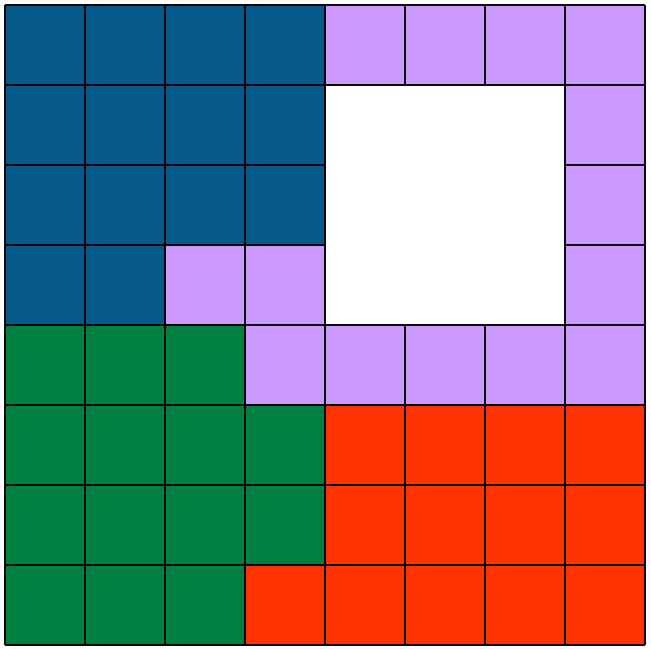
- Try a cyclic distribution
  - load: 12,14,14,15
  - boundary: 12+14+14+15=55
- Terrible communications!
  - want load=14,14,14,13
  - with minimum comms



- **How do we balance the load intelligently ...**
  - **with a sensible communications load?**
- **Need to use non-square subdomains**

New Decomp

- Statistics
  - load: 14,13,14,14
  - boundary: 12+11+12+14=49
- Note
  - for a real (large) problem, much less of each subdomain would be a boundary



- **Problem**
  - **how do we do this automatically?**
  - **for meshes with millions of cells?**

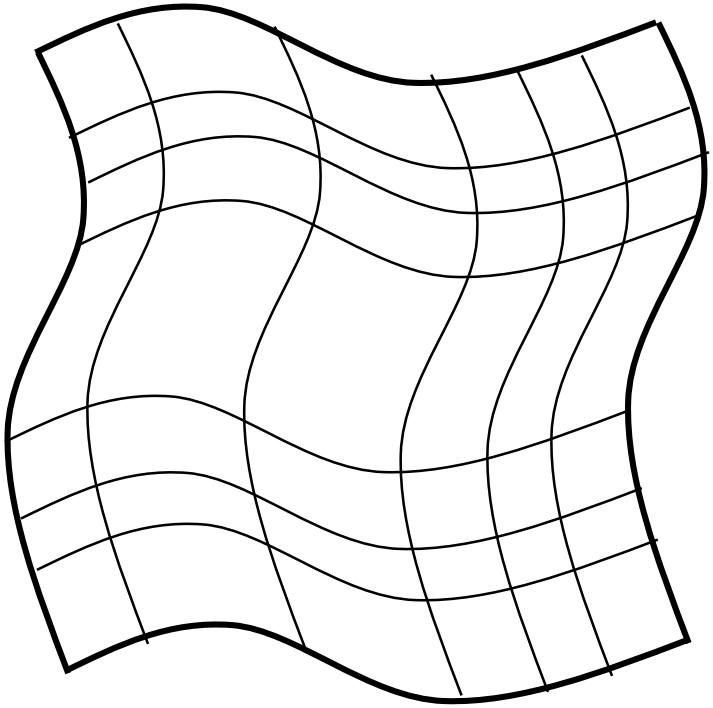
Unstructured Meshes

- Many real calculations cannot be done on regular grids
  - eg complex geometries do not have straight edges
  - in engineering calculations we want to deal with **real** objects
- One standard approach is to use triangles
  - or tetrahedra in three dimensions
  - much easier to fit the mesh to an irregular shape
- When we decompose for parallel computation
  - want same number of triangles in each subdomain
  - minimum number of triangles on subdomain boundaries
- Will not cover how to generate these meshes
  - would be an entire course in itself!

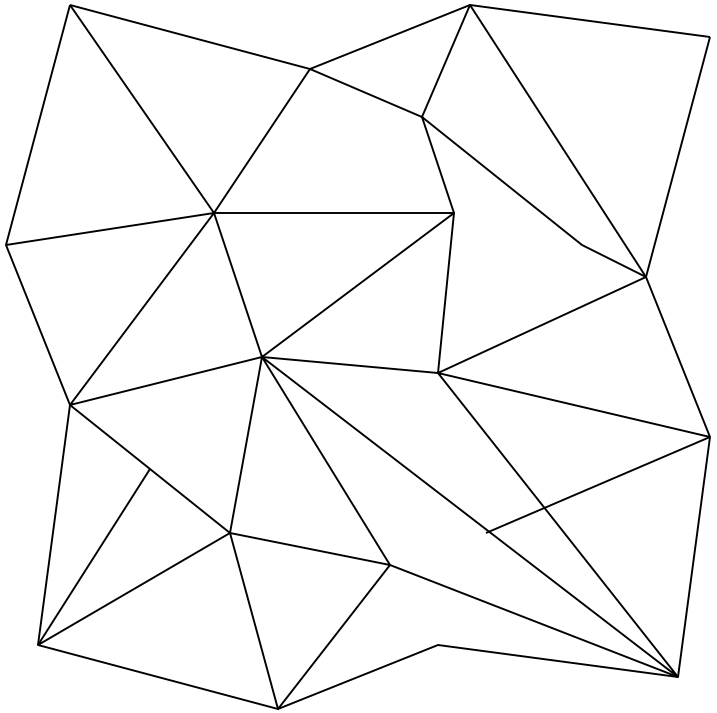
Unstructured Meshes

- How are unstructured meshes distinct from regular grids?
- Regular grids are (topologically) cartesian grids
  - they may be represented by arrays
- An unstructured mesh has no regular structure
  - an element in the mesh may be connected to an arbitrary number of neighbours
  - hence, the mesh cannot be represented by an array
  - a more complex data structure must be used

Examples

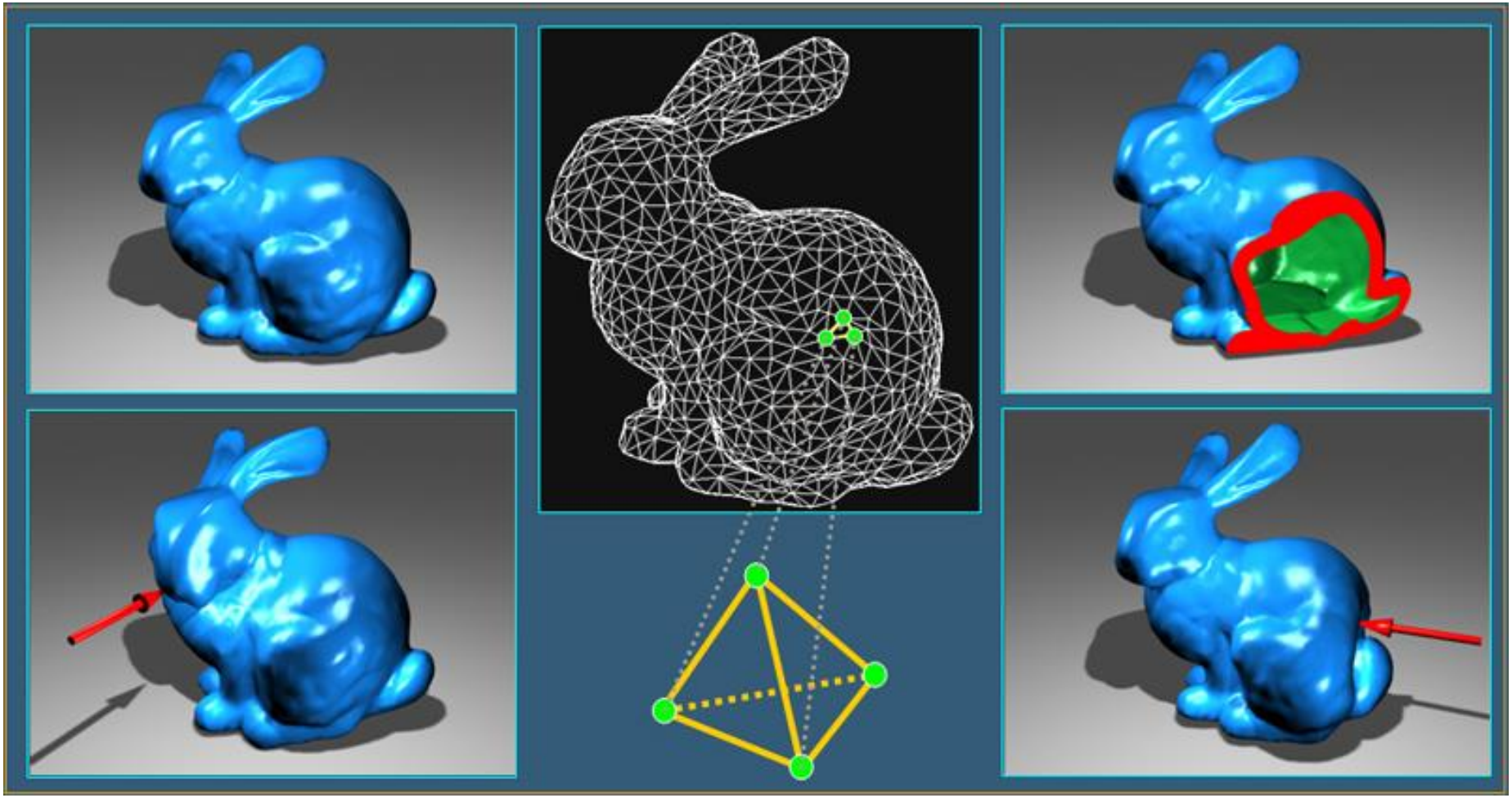


Regular Grid

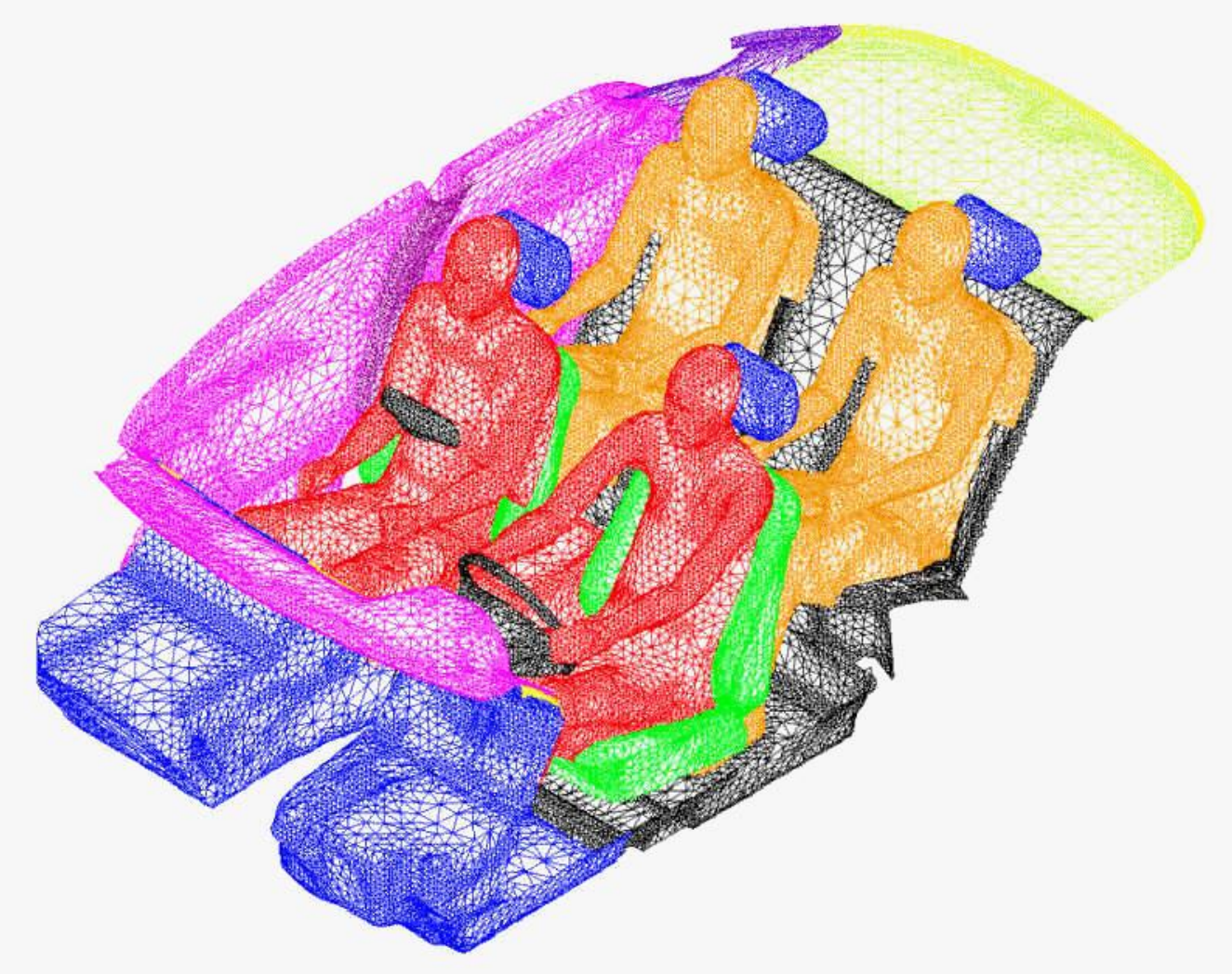


Unstructured Mesh

Example: Visualisation



Example: Crash Simulation



Example: Medical Physics

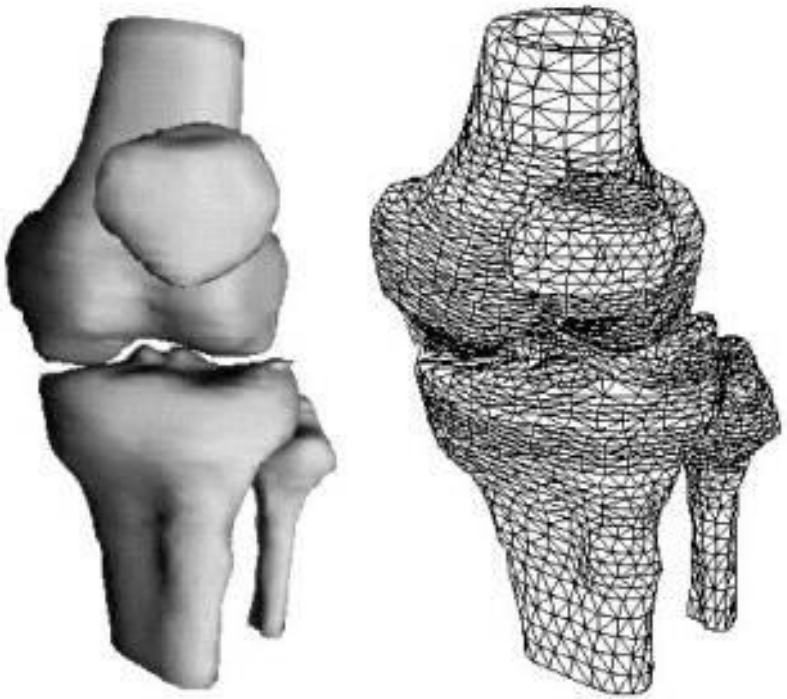


Fig 3: A knee joint consisting of the lower femur, the upper tibia and fibula and patella:
(a) Gouraud shading and (b) tetrahedralization

Storing Unstructured Meshes

- Not a simple grid
  - cannot be stored as two dimensional array `triangle[i][j]`
- Solution
  - give each triangle a unique identifier 1, 2, 3, ..., *N*-1, *N*
  - for every triangle
    - store a list of its nearest neighbours (this list is called a **graph**)
    - store information about its physical coordinates
  - triangle numbering may have nothing to do with their position
    - depends on how the mesh was originally generated

Decomposition (Partitioning)

- Decompose by dividing mesh amongst processors
  - decompose the **domain** into many **subdomains**
- Decomposition has a highly significant effect on performance
  - arriving at a “good” decomposition is a complex task in itself
  - “good” may be problem/architecture dependent
    - eg depends on latency vs bandwidth of target parallel machine
- A wide variety of well-established methods exist
  - several packages/libraries implement of many of these methods
  - major practical difficulty is differences in file formats!

Decomposition Quality

- What makes a “good” decomposition?
- Load balance
  - elements should be distributed evenly across processors, so that each has an equal share of the work
- Communication costs should be minimised
  - there should be as few as possible elements on the boundary of each subdomain, to reduce total volume of communication
  - each subdomain should have as few neighbouring subdomains as possible, to reduce the impact of communications latency
    - ie send as few messages as possible
- Distribution should reflect machine architecture
  - comms/calc and bandwidth/latency ratios need to be considered
    - eg if communications is slow, may accept larger load imbalance
  - e.g. map neighbouring subdomains to neighbouring cores

Problem Complexity

- Graph partitioning has been shown to be *N-P* complete
  - this means that no exact solution may be found in any reasonable time for non-trivial examples
- Certainly complete enumeration is unfeasible
  - the search space is of size P^N, where P (#subdomains) may be in the hundreds and N (#elements in the mesh) in the millions
- We must therefore resort to heuristics which will give us an *acceptable* approximate solution in an *acceptable* time

Practical Methods

- In practice, most decomposition algorithms:
- Impose exact load balance
  - try to minimise boundary length / surface area with this constraint
- Assume an “ideal” homogeneous architecture
  - may not explicitly consider number of neighbouring subdomians
  - do not suggest any mapping of subdomains to cores

Algorithms

- Global methods
  - direct *P*-way partitioning
  - recursive application of some simpler technique
- Local refinement techniques
  - incrementally improve quality of an existing decomposition
- Hybrid techniques
  - using various combinations of above

Global Methods

- Simple techniques
- Random and scattered partitioning
  - very high communication cost
- Linear partitioning
  - regular domain decomposition for unstructured meshes
  - for a mesh of *N* elements on *P* processors give the first *N/P* elements to the first subdomain, second *N/P* to second subdomain, etc ...
  - can give good results due to data locality in element numbering
  - … but may give terrible results!

Global Methods

- Recursive partitioning
- Rather than directly arriving at a *P*-way partition
  - recursively apply some *k*-way technique, where $k << P$
  - typically this means recursive *bisection* of the mesh (*k*=2)
  - *quadrisection* (*k*=4) and *octasection* (*k*=8) may also be employed
  - the latter, and higher order methods, are sometimes referred to as *multi-dimensional* methods
- Apply same criteria *separately* at *each stage* of recursion
  - load balance
  - minimisation of boundary size

Global Geometry-Based Methods

- Geometry based recursive algorithms
  - in most physical problems we have coordinate information for each node in the mesh
  - *ie*, information about physical geometry
- Can exploit this information for mesh decomposition
  - coordinate partitioning
  - inertial partitioning

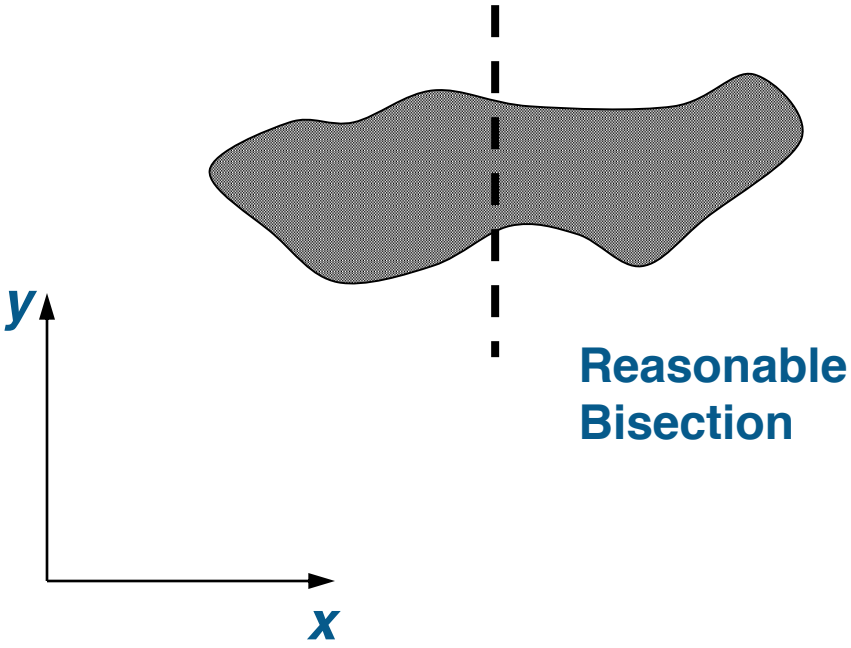
Coordinate partitioning

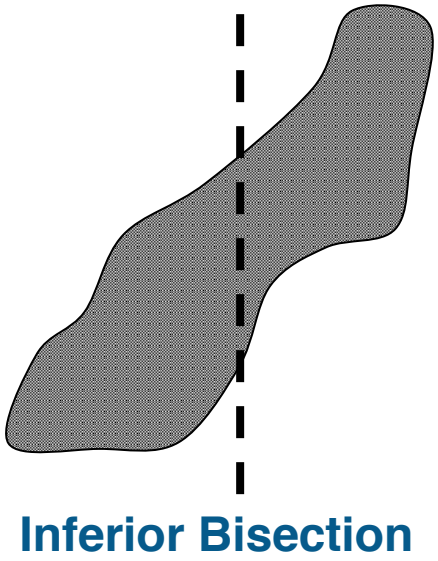
- Compute coordinates of centre of each element
  - which coordinate is used is determined by the longest extent of the domain ie, the *x*-, *y*- or *z*-direction
  - mesh is recursively bisected based on median coordinate value

- Fast and simple to implement method, but
  - can lead to subdomains which are not connected (not surprising given that it takes no account of mesh connectivity information)
  - also suffers if the simulation domain is not aligned with any of the coordinate directions

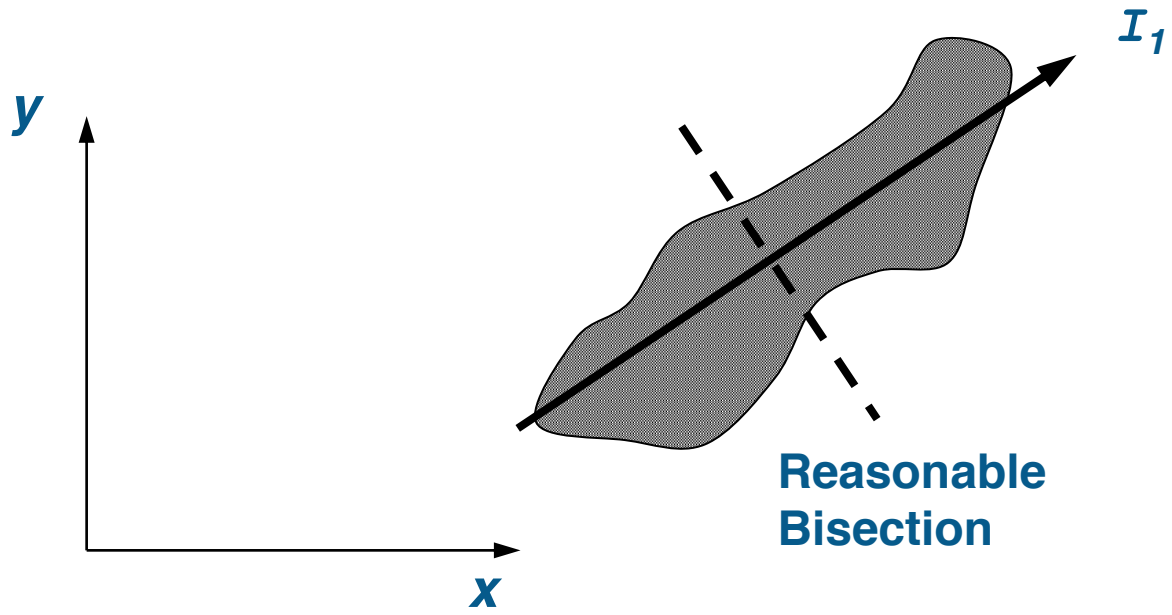
Global Methods

- Coordinate partitioning
- Restriction to *x*-, *y*- or *z*-planes may be inappropriate





- **Inertial partitioning**
- **Project onto the preferred axis of rotation of domain**



Global Methods

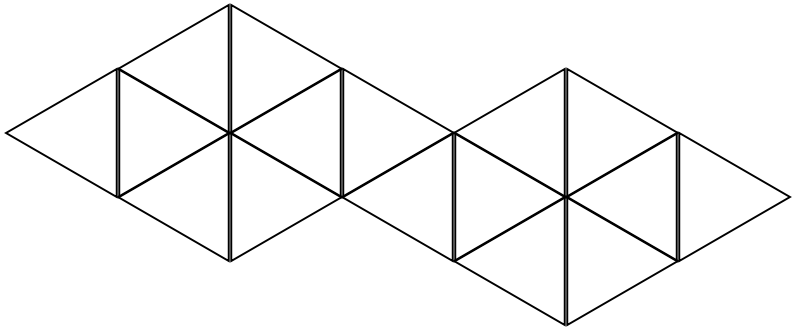
- **Inertial partitioning**
- **Features of inertial partitioning**
  - **quality is on the whole good ...**
  - **... but may be poor in terms of local detail**
  - **no attempt made to ensure that subdomains are connected**
  - **a fast algorithm, due to its relative simplicity**
- **Can form the basis for a competitive strategy**
  - ***eg*, use in combination with a local refinement technique**

Dual Graphs

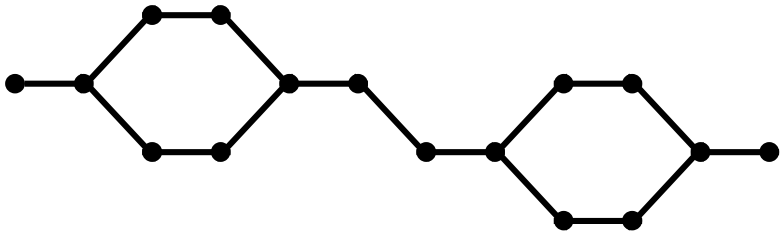
- To make a better decomposition we need
  - a representation of the basic elements being distributed (*eg*, the triangular elements in the case of a finite element mesh)
  - an idea of how communication takes place between them
- A *dual graph*, based on the mesh, fills this role
  - *Vertices* in the graph represent the elements
  - *Edges* in the graph represent transfer of data
    - which may lead to communication in a parallel program
- Graph depends on how data is transferred
  - for meshes it could be via nodes, edges or faces, so ...
  - ... a single mesh can have more than one dual graph

Dual Graphs

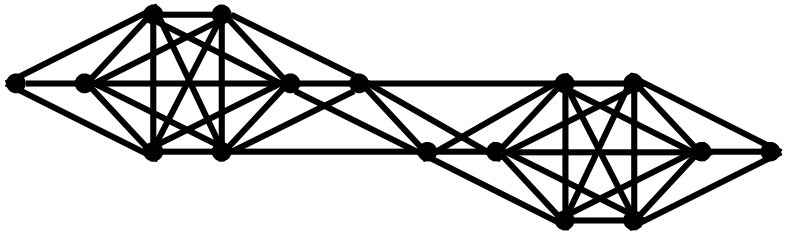
Mesh



Edge-based Dual Graph



Node-based Dual Graph

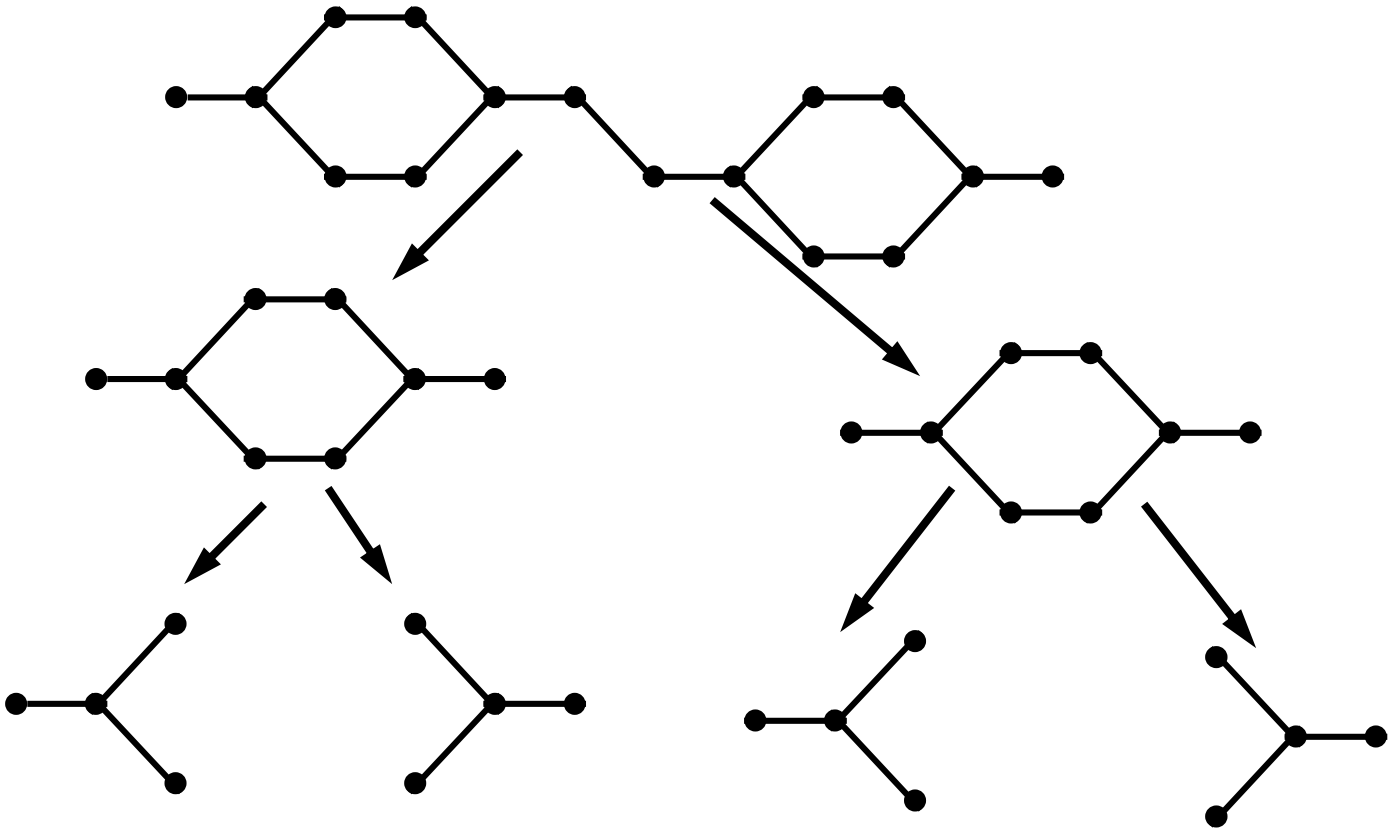


Decomposing with the Graph

- Divide the graph into subsets
  - one subset for each of the subdomains (ie for each of the *P* processors in a parallel program)
- A decomposition of the mesh is a mapping of each of the vertices of the graph to one of the *P* subsets
  - load balance means an equal number of vertices in each subset
- What about communications?
  - we count the number of *cut edges*
  - ie number of edges that connect vertices that are in different subsets
- Graph analysis a major topic in classical Computer Science

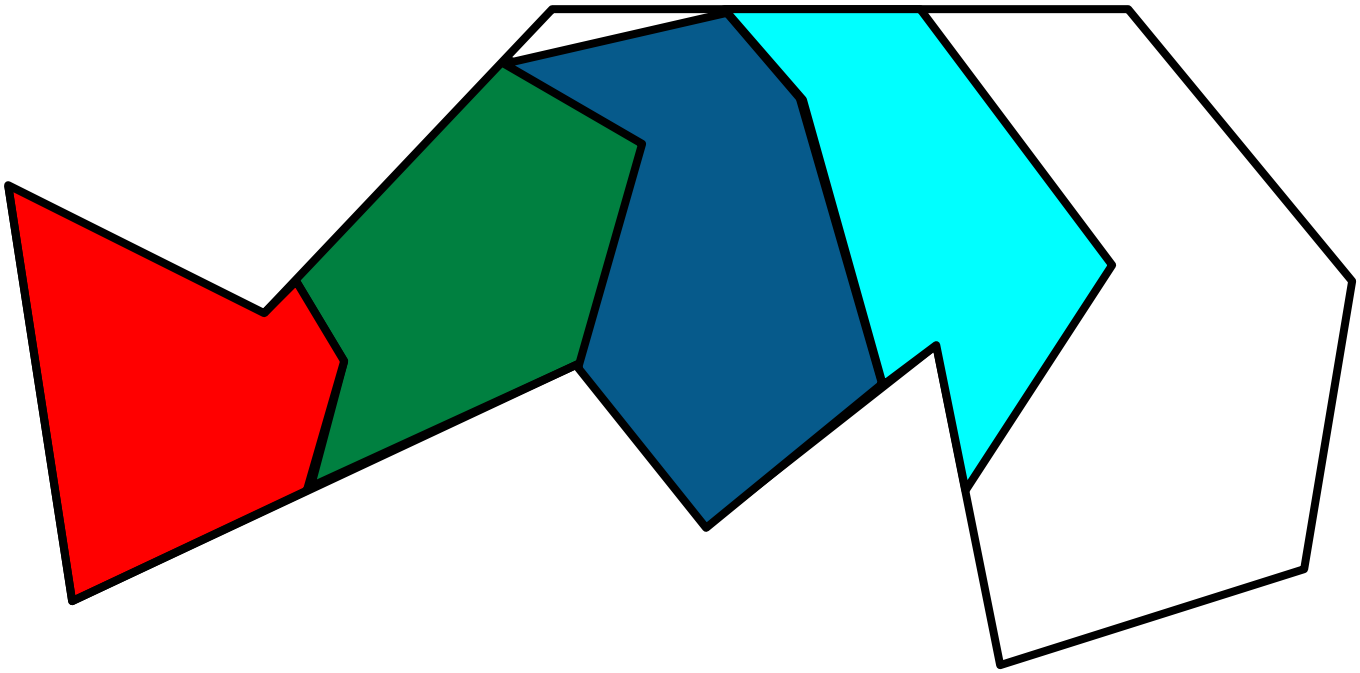
Graph decomposition

- e.g. recursive bisection leading to 5 cut edges



Greedy Algorithm

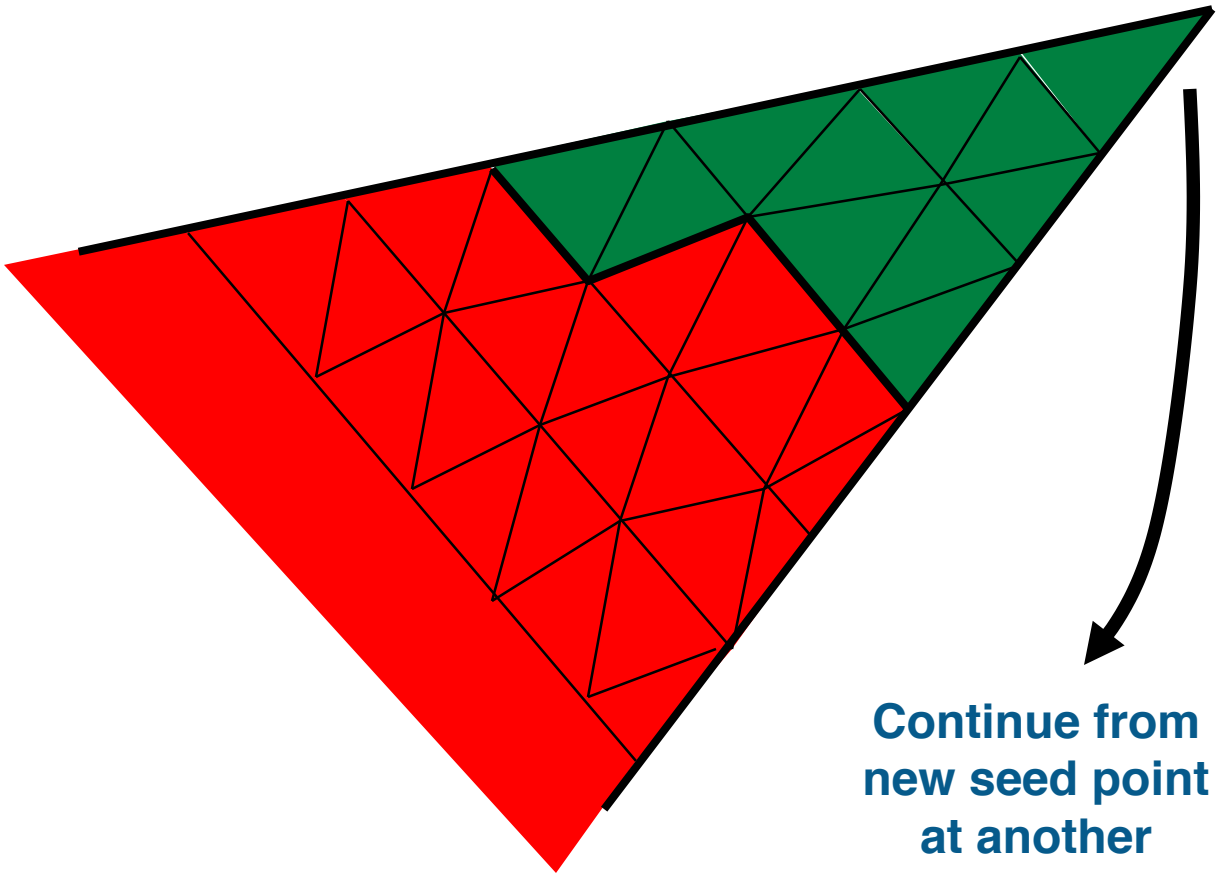
- Bite successive chunks out of the mesh
  - take bites of the correct size to ensure load balance



(My) Greedy Implementation

- Works purely on dual graph
  - only concerned about connectivity
- Works by expanding in shells
  - find all nodes on the boundary shell of current (incomplete) subdomain
    - add these nodes to the subdomain
  - find the boundary shell of the new larger subdomain
    - stop when subdomain is of correct size
- Need a seed point to start each subdomain
  - take first seed at a corner (node with fewest neighbours)
  - when a subdomain is complete, use next shell point as new seed
- What if there is no shell (a dead-end in the graph)?
  - continue from a new seed point chosen at a new corner
- My method does not always generate the same decomposition
  - if there is more than one possibility for a seed, choose at random

Adding Shells to Subdomains

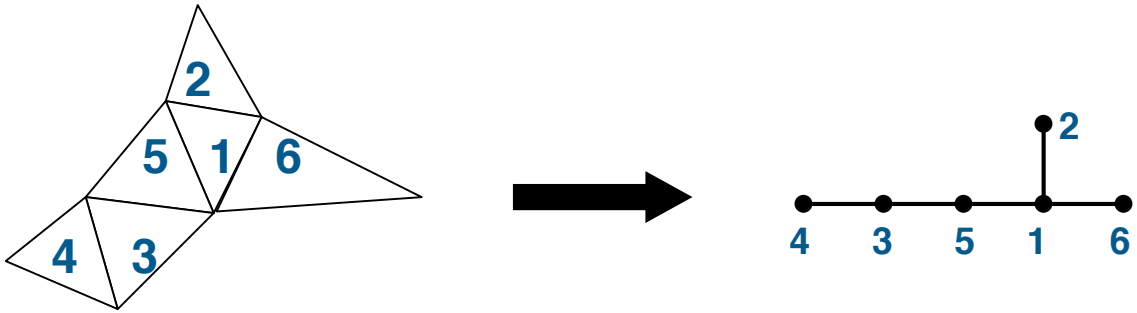


Global Methods

- Greedy algorithm
- Advantages
  - it is a very fast algorithm
  - directly yields the required number of partitions
  - load balance is even
  - subdomains generally have good aspect ratios
- Disadvantages
  - often generates disconnected subdomains
  - this may lead to high communication cost

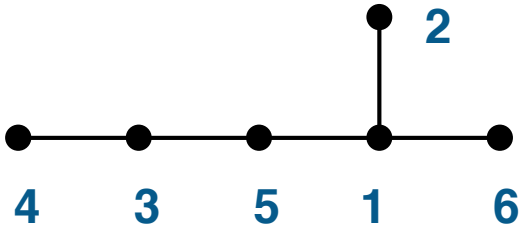
Spectral Partitioning

- Sophisticated but computationally expensive
- First consider this mesh and graph



The Laplacian Matrix

- Represent dual graph as a sparse, symmetric matrix
- To obtain *NxN* Laplacian matrix *L* of graph with *N* vertices
  - L_{ij} = -1 if vertices *i* and *j* are connected by an edge, 0 otherwise
  - L_{ii} = the number of neighbours of vertex *i*



$$\begin{pmatrix} 3 & -1 & & & -1 & -1 \\ -1 & 1 & & & & \\ & & 2 & -1 & -1 & \\ & & -1 & 1 & & \\ -1 & & -1 & & 2 & \\ -1 & & & & & 1 \end{pmatrix}$$

Spectral partitioning

- Bisect based on values of the *eigenvectors* of *L*
  - compute eigenvector corresponding to second smallest eigenvalue
  - use this vector (the *Fiedler* vector) as the *separator field*
  - use the median value to split graph into two parts
- One of the most popular and widely used methods
  - generates very good overall decompositions
  - generally superior to greedy, coordinate and inertial methods
  - local deficiencies may be tidied up, giving excellent final quality
- But can take a lot of time and memory
  - computing eigenvectors is an extremely expensive calculation
  - the iterative Lanczos algorithm is often used for eigensolution

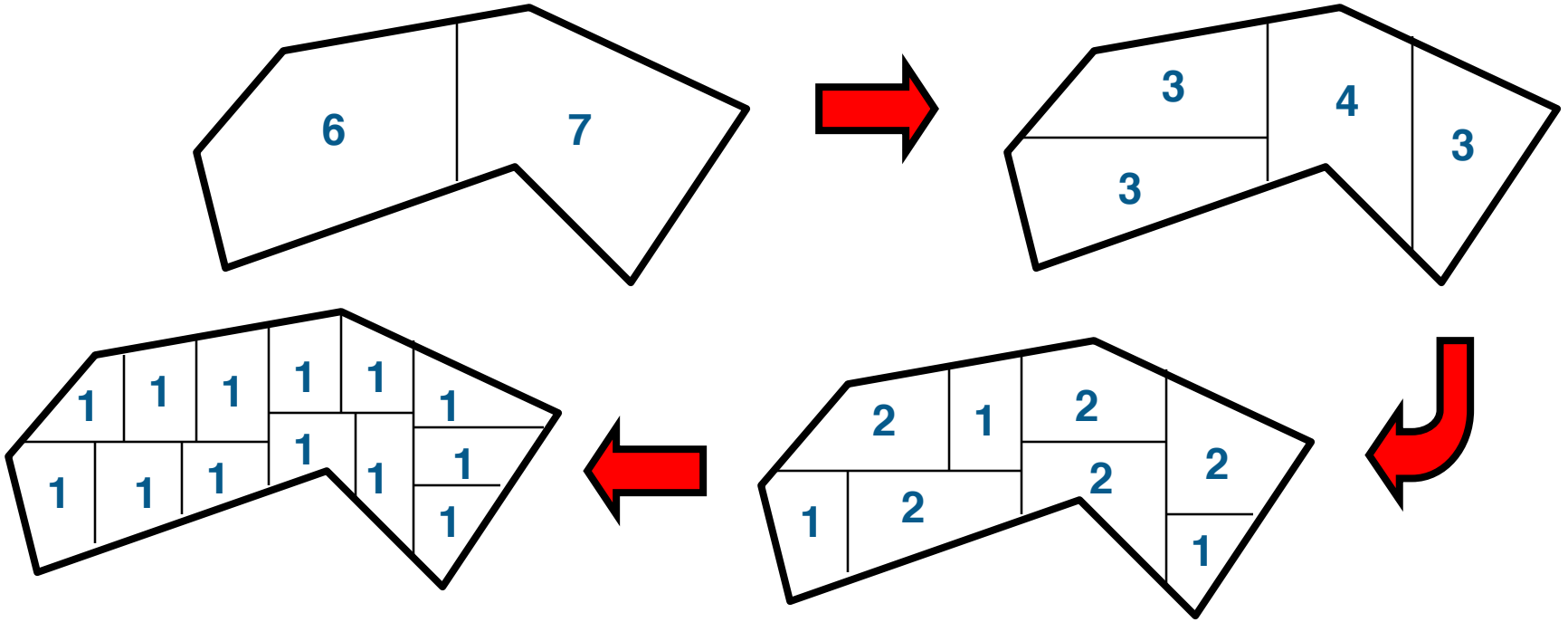
Spectral Partitioning

- You will see that it is a good decomposition
  - but what is the mathematical justification?

- Too complex to cover here, but the basic outline for bisection is:
  - formulate mathematically as a minimisation problem
    - try to miminise the number of cut minus uncut edges
  - each vertex is assigned to one of two subdomains: +1 or -1
    - decomposition *x* is a vector of length *N*, eg (+1, +1, -1, -1, +1, -1, ...)
    - ie each decomposition is a different corner of an *N*-dimensional hypercube
    - try to find the solution *x* that minimises $H(x) = x^T L x$
    - *H(x)* counts the number of cut edges
  - cannot solve this discrete problem exactly, so
    - map to a continuous problem where *x* is a real vector: $-1 <= x_i <= +1$
    - fix length of *x* so solution is on the surface of an *N*-dimensional *sphere*
  - solution of continuous problem is vector *x* with second-smallest eigenvalue
  - map back to the discrete solution by finding closest point on hypercube to the solution on the surface of the sphere

Non Power-of-two Subdomains

- Can extend bisection for arbitrary number of domains
  - at each stage, still divide subdomains into two smaller parts
  - parts are no longer equal; weight sizes as appropriate
- For example, for 13 subdomains with RCB:
  - recursive co-ordinate bisection



Local Refinement

- Kernighan and Lin
- If we consider two adjacent subdomains, S_a and S_b, we compute a gain associated with moving a vertex from its current subset to the other:
  - the gain is simply the reduction in the number of cut edges resulting from the swap
  - swap may make things worse, in which case the gain will be negative
- Could just make all swaps with positive gain
  - but this would get trapped in the first local minimum it encounters
- The Kernighan and Lin approach avoids this by also considering swaps which make matters worse

Local Refinement

- Kernighan and Lin

- The algorithm does not find very good partitions of large graphs unless it is given a reasonable starting point
- It is therefore best used in conjunction with a cheap global method, which provides a reasonable overall partition but may be poor in its details
  - eg, recursive inertial partitioning

Hybrid methods

- Combinations of Techniques
- Features of global methods
  - give a reasonable decomposition when viewed on a large scale
  - on a small scale they are often lacking in quality
- This can often be fixed using local refinement
- For example
  - spectral decomposition may be better than inertial
  - when inertial is coupled with KL the result is superior to pure spectral and may be faster to compute
  - coupling spectral with KL produces a better result still

Summary

- Unstructured meshes are important for a wide class of problems
  - but their decomposition onto parallel machines is non-trivial
- Range of different heuristic algorithms developed
  - *global methods*: greedy, recursive partitioning (coordinate, inertial, spectral), ...
  - *local refinements*: kernighan & lin, jostle, ...
- Techniques can be combined effectively
- Useful for various awkward geometries
  - best method is Recursive Spectral Bisection (RSB)
  - expensive in time and memory